Hyperion: Building the largest in-memory search tree

MARKUS MÄSKER, Johannes Gutenberg University Mainz
TIM SÜSS, Johannes Gutenberg University Mainz
LARS NAGEL, Loughborough University
LINGFANG ZENG, Huazhong University of Science and Technology
ANDRÉ BRINKMANN, Johannes Gutenberg University Mainz

Indexes are essential in data management systems to increase the speed of data retrievals. Widespread data structures to provide fast and memory-efficient indexes are prefix tries. Implementations like *Judy*, *ART*, or *HOT* optimize their internal alignments for cache and vector unit efficiency. While these measures usually improve the performance substantially, they can have a negative impact on memory efficiency.

In this paper we present Hyperion, a trie-based main-memory key-value store achieving extreme space efficiency. In contrast to other data structures, Hyperion does not depend on CPU vector units, but scans the data structure linearly. Combined with a custom memory allocator, Hyperion accomplishes a remarkable data density while achieving a competitive point query and an exceptional range query performance. Hyperion can significantly reduce the index memory footprint, while being at least two times better concerning the performance to memory ratio compared to the best implemented alternative strategies for randomized string data sets.

1 INTRODUCTION

The amount of data being generated and processed has grown dramatically. The Internet, social media, sensors, and logs produce so much information that traditional processing and storage technologies are unable to cope with it. These data are often accessed under near-real-time requirements, e.g., when companies like Twitter or Facebook dynamically provide web pages to their users [40][47]. In-memory database clusters based on, e.g., Redis or Memcached are here used as caches to fetch thousands of small key-value (k/v) pairs for assembling a single web page [31][12]. Data and metadata of such caches must be highly compressed to use memory efficiently, and well-structured indexes are required to guarantee low access latencies.

This paper investigates indexing data structures for in-memory databases with respect to the trade-off between performance and memory consumption and presents *Hyperion*, a trie-based indexing key-value store designed with memory efficiency being the main objective.

Search trees have been used to index large data sets since the 1950s [19]; examples are *binary search trees* [29], *AVL trees* [1], *red-black trees* [8], *B-trees* [9] and *B+-trees*. These trees have the disadvantage of always storing complete keys inside their nodes and hence large parts of the keys many times. Binary trees also show bad caching behavior and do not adapt well to new hardware architectures [44]. The resulting performance issues become worse due to the increasing divergence between CPU and memory speed [42][46].

More memory-efficient data structures are *tries* [19][21] which distribute the individual parts of a key over multiple nodes to reduce redundancies. A key is reconstructed by following a path in the trie and concatenating the individual parts stored in the nodes. Since all keys of a sub-trie share a common prefix, tries are also called *prefix tries* or *radix tries*. Tries are not balanced, and their lookup complexity will grow linearly in $O(n)$ with the key size if the paths are not compressed. Tries are nevertheless faster than less memory-efficient B+-Trees (even when adapted to new architectures [45][25][32]) in many scenarios [35][5][3][10].

Memory is one of the most expensive server components. Indeed, memory accounted for more than half of the costs of the servers used as evaluation platform for this paper. Recent studies have shown that the individual key and value sizes within k/v stores in industrial settings are rather small, e.g., values smaller than 500 bytes account for more than 80% of the overall capacity within most of Facebook's key-value stores [6]. Although tries tend to be more memory-efficient than trees, trie-based indexes are still responsible for taking large parts of the main memory [48], and it remains a challenge to further reduce their metadata overhead.

Several optimization strategies were explored [27][7][5][11][33][35], but analyzing their design and comparing their benchmark results, it seems that these tries were still constructed with performance as the primary objective. Hyperion, on the other hand, focuses on handling huge indexes in a very memory-efficient way.

Performance is not neglected though, Hyperion tries to find a sweet-spot between memory consumption and performance. The number of requests processed by each server in scale-out scenarios is determined by CPU utilization *and*

Authors' addresses: Markus Mäsker, Johannes Gutenberg University Mainz, maesker@uni-mainz.de; Tim Süß, Johannes Gutenberg University Mainz, suess@uni-mainz.de; Lars Nagel, Loughborough University, l.nagel@lboro.ac.uk; Lingfang Zeng, Huazhong University of Science and Technology, lfzeng@hust.edu.cn; André Brinkmann, Johannes Gutenberg University Mainz, brinkman@uni-mainz.de.

network saturation (see Twitter use case [47]) and even 10 GBit/s Ethernet is unable to handle more than a few million messages per second. The performance objective of Hyperion is therefore to be fast enough to not become the bottleneck of distributed in-memory k/v stores, while k/v stores in such scale-out environments are Hyperion's first use case.

Thanks to its extreme data density, Hyperion can therefore also be used in Internet-of-Things (IoT) scenarios, where many data streams have to be online processed in edge devices by embedded systems with limited processing and memory capacities [38]. This includes, e.g., traffic time series for network monitoring purposes [34] or storing and processing machine logs with self-similar entries [26]. Additionally, Hyperion is able to efficiently store potentially arbitrarily long keys, becoming necessary, e.g., for future DNA sequencing techniques providing longer base sequences [13].

We analyze Hyperion and four state-of-the-art tries regarding point and range queries and memory consumption: Judy Arrays [7], the HAT-trie [4], the Adaptive Radix Trie (ART) [35], and the Height Optimized Trie (HOT) [10]. A red-black tree (STL `map`) and a hash table (STL `unordered_map`) are included as baselines. The data structures are evaluated as k/v stores using integer keys with sparse and dense distributions and a real-life string data set [24].

Our main contributions are:

- Hyperion, an indexing data structure that is twice as memory efficient for string data sets than all existing competing data structures and that provides exceptional (and best) range-query performance, while furthermore still being more than 20% more memory efficient than potential optimizations of other trie data structures.
- An evaluation of indexing structures of unprecedented size within single servers. Previous evaluations [4][5][35][3] only considered subsets of the key types, operations, and structures covered here. The new results highlight individual strengths and indicate that some previous results cannot be extrapolated to huge data sets, while also showing the potential of HOT and ART if they would be transformed into k/v stores.

Hyperion does not yet implement fine-grained thread-parallelism, but it already allows to split tries in up to 256 separately locked and thread-safe arenas. They are not optimized yet and only provide limited speed-ups. It is nevertheless already well-suited for huge simulations involving several billion index elements within a single server or as a building block for scale-out in-memory databases.

2 BACKGROUND

The concept of a trie grew out of the requirement to index byte sequences of variable length [19]. The term *trie* was introduced as a means to store and re*trie*ve information that is referenced by keys of variable length [21]. This section provides definitions and an overview of existing tries describing the ones most relevant to our work in more detail.

2.1 Definitions

A *trie* or *prefix tree* is a rooted search tree that is commonly used for storing vectors of characters which we usually refer to as *(indexed) keys*. Every node, or alternatively every edge, is attributed with a character of an underlying alphabet. A path from the root to any node represents a string, namely the concatenation of the characters. Figure 1 shows a trie storing the words (or keys) *a*, *and*, *be*, *that*, *the*, and *to*. The root node represents the empty string and nodes that are grayed ends of words.

Morrison [39] reduced the memory footprint of a trie by using *path compression*. If a key suffix forms a path without branches, then the suffix can be stored as a string in a single leaf node. Generally, if a node in a trie has only one child, then parent and child can be merged by concatenating their characters. A *radix tree* (also known as *radix trie* or *patricia trie*) is such a compressed trie. In the remainder of the paper we refer to radix trees also as tries.

In a *binary trie*, every node represents a single bit; e.g., 0 for a left child, 1 for a right child. An *m-ary trie* is a trie where each node has up to m children and encodes $\lceil log_2(m) \rceil$ bits of the key. A key K of n bits can be defined as a sequence of such *partial keys* k_i:

$$K = k_0, k_1, ..., k_{(\frac{n}{\lceil \log_2(m) \rceil})-1} \tag{1}$$

Since each k_i is stored in one trie node, the height of the trie is $O(\frac{n}{\lceil \log_2(m) \rceil})$. Accessing a node usually results in at least one cache line fill and the trie height is a lower bound for the expected number of cache line fills per request. Increasing m leads to a higher fan-out and hence a reduction of the trie height and number of cache line fills. At the same time, it increases the size of the nodes, leading to more complex partial key comparisons. Especially for sparsely populated nodes a large fan-out can be inefficient.

Most tries store keys in lexicographical order, which might not be the natural order of the data. We use the transformations from Leis et al. [35] to obtain binary-comparable keys.

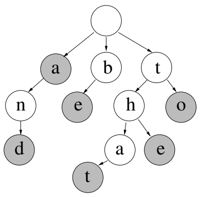


Fig. 1. A trie representation of English words

2.2 Judy arrays, HAT-trie, ART, and HOT

Judy arrays were among the first data structures that dynamically adapt the memory usage of a node to the actual number of keys [7][3]. Judy arrays are 256-ary radix trees and come in three different flavors: *Judy1* is an associative array that maps integer keys to boolean values and is used to create sets. *JudyL* is an int-to-int map, and *JudySL* maps strings of arbitrary length to int values. JudyL arrays form the inner nodes of JudySL tries.

Judy distinguishes *uncompressed nodes* and two types of compressed nodes, namely *linear* and *bitmap nodes*. The internal data structures are designed to minimize the number of cache line fills per request. Judy also applies various compression techniques, which can be grouped into two categories: *horizontal* and *vertical compression*. While *horizontal compression* aims to decrease the size of internal nodes, *vertical compression* reduces the number of nodes.

The *burst trie* [27] was designed to store string keys of characters chosen from an alphabet Σ. A burst trie consists of *trie nodes* and *containers*. A trie node represents one character and keeps an array of size $|\Sigma|$ with pointers to the next nodes or containers. Since lower trie levels tend to be sparsely populated, the burst trie merges small enough sub-tries into a container. Once a container becomes too large, it *bursts* and is replaced by a trie node and new, smaller containers. The authors examined different heuristics for bursting and tested linked lists, binary trees and splay trees for managing partial keys in a container. The *HAT-trie* [4] is a successor of the burst trie and manages the partial keys of a container in a cache-conscious hash table. Hash tables allow fast access rates, but complicate range queries.

The *Adaptive Radix Tree* (ART) is a cache-line-optimized 256-ary trie [35]. It uses four node types to reduce memory overhead and improve cache utilization: *Node4*, *Node16*, *Node48*, and *Node256*. The number specifies the maximum number of entries. Each node has a 16 byte header specifying the node type, child count and a compressed path. The header of a Node4 or Node16 is followed by two arrays, a `char` array containing the stored key characters, and a `pointer` array referencing the children. Node4 elements are compared iteratively, but Node16 elements using SIMD instructions. A Node48 has a 256-elements char array and an array of 48 child pointers. The first array stores the index of the corresponding elements in the pointer array. The key defines the index in the character array and the retrieved entry defines the index in the pointer array if present. Finally, a Node256 node is a 256-element child pointer array.

The *height optimized trie (HOT)* is a generalized Patricia trie that is able to adapt the number of bit m considered in each node by combining multiple nodes of a binary Patricia trie into compound nodes [10]. It is therefore able to adapt the span of its nodes to the key distribution such that each node has a high fan-out independent of the data set. HOT supports range queries and its node layout allows to use SIMD-instructions, while the fine-grained node locking enables concurrent put and get operations. The adaptability of the fan-out also leads to more densely populated nodes, where presented memory savings compared to ART can be a factor of three.

2.3 Other Tries

The *C-trie* [36] is an m-ary trie, which is optimized for read-only workloads. Nodes of the trie keep an m-sized bitfield to indicate the existence of a child node. All nodes are enumerated and serialized into a single bit stream which is why updates can cause a complete reconstruction of the trie.

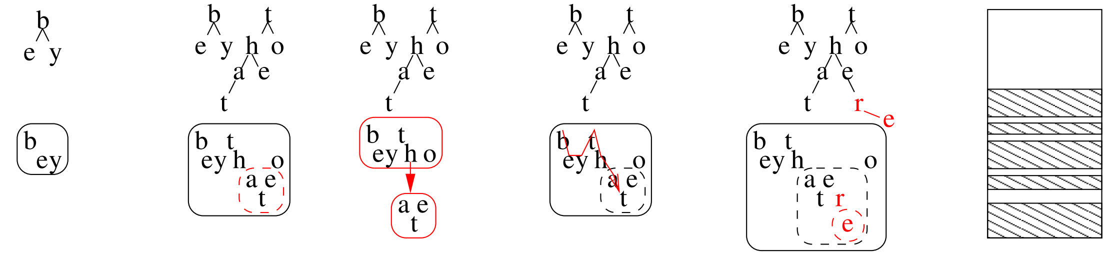
The *generalized prefix tree* (GPT) stores its trie nodes in huge pre-allocated memory segments [11]. Nodes are referenced by their offset within the segment and the base memory pointer of the segment. GPT reduces the storage costs of memory allocation and heap fragmentation and, at the same time, reduces the memory footprint of child node pointers. But since GPT uses neither path compression nor adaptive node sizes, other tries such as ART provide a better worst-case memory efficiency [35].

The *KISS-Tree* [33] is a specialization of the GPT and uses 32 bit keys. They are split into three sections, f_1(16 bits), f_2(10 bits) and f_3(6 bits), generating three levels. On the first level, no memory lookups are necessary because the second-level address is directly computed using f_1. Direct addressing requires that the nodes of level two are sequentially stored in memory. The bits of f_2 then identify one of 1,024 buckets on the third level each containing a *compact pointer* (i.e., a 64 bit pointer reduced to 32 bit). The last six bits address a bucket of the selected third-level node. On this level, the nodes are compressed, and a 64 bit map indicates which nodes exist.

3 HYPERION

Memory-efficient trees must have a high information density per node, while keeping the total number of nodes small. Tries naturally achieve a high information density by reducing redundancy when storing common prefixes only once. Having fewer nodes reduces overhead and memory fragmentation. However, increasing the node size can be detrimental to the performance as it complicates updates.

Hyperion is based on three main ideas: Reduce the overhead of nodes by increasing the partial key size, decrease internal fragmentation by an exact-fit approach, and optimize memory management. Hyperion is therefore an *m-ary trie* with $m = 65,536$ being large compared to other tries. Each node of the trie encodes up to 16 bits of the overall key, leading to fewer, but bigger nodes. Hyperion stores nodes in containers and adapts the size of containers to their actual



(a) Basic container exact−fit layout

(b) Embedded container

(c) Container ejection

(d) Linear search key lookup

(e) Order preserving key insertion

(f) Custom memory management

Fig. 2. Hyperion core concepts. The red arrow in (d) shows the linear lookup steps for the key *"that"*.

population, applying an exact-fit approach. It is therefore feasible to encode the 16 bit partial keys into containers without excessive internal fragmentation.

Figure 2 demonstrates Hyperion's core ideas. Figure 2a shows a sketch of a basic container storing the 16 bit keys *"be"* and *"by"*. A container splits them into two 8 bit keys (visualized by the vertical offset) which allows to store the *b* only once. The exact-fit layout of the node is illustrated by the tight frame around the keys.

Figure 2b shows a larger example graph adding the keys *that*, *the*, and *to*. Unlike ART or HAT, Hyperion does not rely on fixed offsets and is able to recursively embed small containers into their parents, reducing the number of containers in the trie and improving cache locality.

Although large containers are more memory-efficient, they cannot grow indefinitely to keep costs for updates and searches small. Once an embedded container becomes too large, it is ejected by its parent (see Figure 2c).

Sketches of the key lookup and insertion procedure are shown in Figures 2d and 2e. Lookups perform a pre-order traversal of the container's data structure. The linear scanning procedure is slower than SIMD operations or direct offset calculations but also not constrained to static offsets and fixed sizes. The implementation of the data structure is therefore highly compact and flexible. Maintaining order among the keys is crucial for achieving fast lookups and range queries. These methods and other performance and memory efficiency considerations are presented in Section 3.3.

A compact trie data structure alone does not guarantee memory efficiency. Hyperion's containers grow in small 32 byte increments to keep over-allocation low. This growth pattern is very unusual and can lead to excessive memory fragmentation. Since the pattern is also very predictable, we built a custom memory manager that is tailored to our use case. It prevents excessive *external fragmentation* and introduces the *Hyperion Pointer (HP)*, a 5 byte representative stored in our trie instead of 8 byte pointers. This further increases the information density and fully decouples the data structure from its location in memory (see Section 3.2).

Finally, we present the optional *key pre-processing* in Section 3.4. Data structures have to provide a good trade-off between performance and memory consumption. Optimizations for different use cases can be based on providing different implementations or on adapting the data sets themselves. Hyperion's approach is to transparently adapt the data set so that it suits our data structure better.

3.1 Trie Data Structure

Hyperion is designed as a carefully growing 65,536-ary trie. Every node of the trie forms a container and represents a 16 bit partial key which, in the case of internal nodes, is used to identify up to 2^{16} children. The large branching factor results in fewer, but larger nodes.

First, we will explain the container layout and its partial key handling. Once we are familiar with the structure of internal nodes within a container, we will discuss the child-container mechanisms.

Each container has a 4 byte *header* and an appended payload (Figure 3). The header's first 19 bits store the container size and the next 8 bits the number of unused bytes at the end of the container. They are followed by three *jump table* bits denoted by *'J'* and finally by the two bits of the *split container* flag *'S'*. Jump tables and container splits enhance the performance and are discussed in Section 3.3. Containers are initialized with 32 bytes, i.e. with a payload of 28 bytes.

Figure 4 maps the sample trie of Figure 1 onto containers *C1*, *C2*, and *C3*. Containers maintain internal tries to handle the 16 bit partial keys as separate 8 bit keys:

$$k_i = k_i^0 k_i^1 \tag{2}$$

The key *"be"*, e.g., is split into $k_i^0 = b$ and $k_i^1 = e$. We use two internal node types called *"T-Node" (top)* and *"S-Node" (sub)* for k_i^0 and k_i^1, respectively.

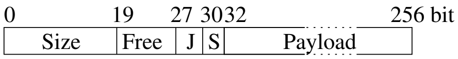


Fig. 3. Container: header and appended payload

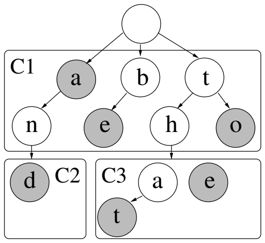


Fig. 4. Containers mapped onto the sample trie

The bit structures of the T- and S-Nodes are shown in Figure 5. The partial key index *'k'* which is the third bit, distinguishes T- from S-Nodes. Its value is equal to the superscript index of Equation (2). The character a and e in container *C3*, e.g., have partial key index `0`, while the character t has index 1. In a container all T-Nodes are siblings, every S-Node is a child of a T-Node, and two S-Nodes are siblings if they have the same parent T-Node.

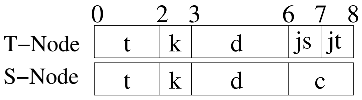
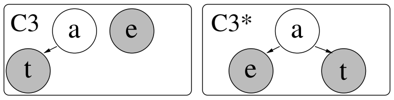


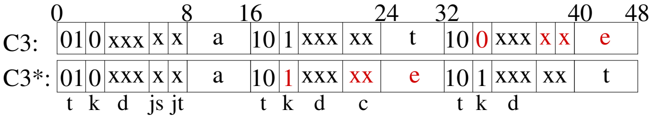
Fig. 5. Bit structure of the T/S-Nodes

The first two bits of a T- or S-Node form the type flag *'t'*, which defines the node type: `01` denotes an *inner* node, `10` and `11` a leaf node, and `00` an *invalid* node. a in container *C3*, e.g., is an inner node; t and e are leaf nodes. Depending on whether its key maps to a value, the type of a leaf is either `10` (w/o value) or `11` (with value). Invalid nodes mark over-allocated memory of a container and correspond to internal fragmentation.

The type and key flags can be explained with the two example containers in Figure 6a. *C3* stores the partial keys at and e whereas *C3** stores at and ae, which makes e and t siblings in *C3**. Containers store their internal tries as byte arrays in pre-order traversal, which means that a T-Node is always followed by its S-Node children. Figure 6b shows a simplified representation of the byte arrays generated for *C3* and *C3**. The container headers are omitted and the yet unexplained flags d, js, jt, and c are replaced with *don't cares* x. The numbers above the arrays are bit offsets.



(a) Graph representation



(b) Byte array representation

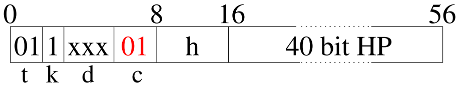
Fig. 6. Two versions of C3 to illustrate the key flag

We exemplarily parse the byte array of *C3*. The first node is an inner node (`t=01`) and a T-Node (`k=0`). Ignoring the *don't cares*, we continue to read the partial key a. The next 16 bit encode the node storing the partial key t. It is a leaf node without value (`t=10`) and an S-Node (`k=1`). Together with its parent, it forms the key at. The last 16 bit define a second leaf node (`t=10`) with partial key *e*. It is a T-Node and hence a sibling of *a*.

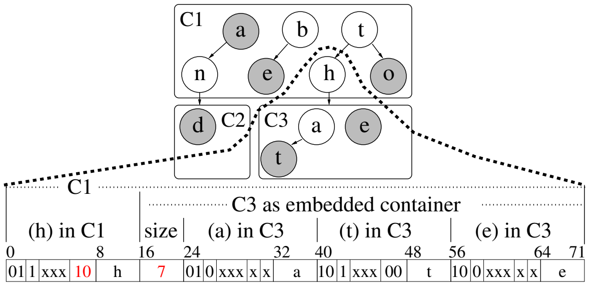
In *C3** the representation of the partial key a is the same. Since e is now a child of a, it is an S-Node, and since e is smaller than t, it must precede t to maintain the order among the siblings. Apart from its position in the byte array, the encoding of the t key is unaltered.

Child Containers: Containers must reference their child containers. T-Nodes have two jump flags (explained in Section 3.3), S-Nodes a two-bit child flag *'c'* (see Figure 5). A `00` child flag indicates that no child container exists. The node is therefore a leaf node which in this case is also redundantly encoded in the previously explained *type* field. A value of `01` declares the existence of a trailing child container pointer. Standard eight byte pointers are replaced by our own five byte *Hyperion Pointer (HPs)* (cf. Section 3.2). Figure 7a shows as an example the byte representation of the h-node in *C1*. This node has a child container (*C3*), hence sets `c=01` and appends a corresponding *HP*.

Figure 7b shows *C3* as an embedded container, which is a special child container that is embedded into the byte array of its parent. The parent *h* defines the existence of an embedded

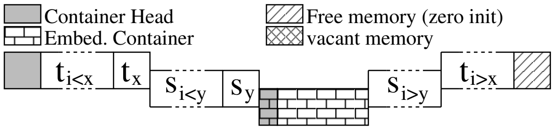


(a) C1's node representing "h" referencing C3 with a Hyperion Pointer (HP)

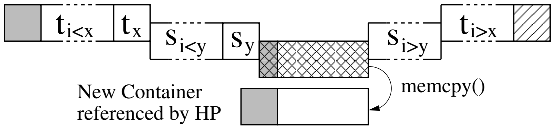


(b) C1's node representing "h" embedding C3

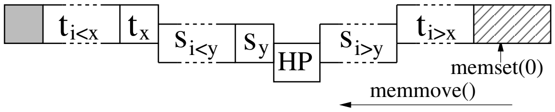
Fig. 7. Byte array representation of the node representing "h" in *C1* with C3 being *referenced* (a) and *embedded* (b)



(a) The initial state before container ejection



(b) After ejection of the embedded container



(c) Final state after updating the old container

Fig. 8. Ejecting an embedded container

container by setting `c=10`. Embedded containers have a one byte header that only stores their total size, which is limited by the maximum size of S-Nodes of 256 bytes. In this case it is 7 bytes including the size field itself. The byte array of the embedded container *C3* is the same as it would be in a normal container (cf. Figure 6b).

Embedded containers must be ejected if they cannot be stored anymore in their parent S-Node (cf. transition of Figure 2b into 2c). Figure 8 shows the necessary steps in more detail. First, the embedded container of the key $t_x s_y$ is ejected (Figure 8b). Memory for the new container is allocated (referenced by the Hyperion Pointer *HP*) and an empty container is initialized. Then the embedded container's payload is copied to the new container and its *size* and *free* fields are updated.

Updating the old container is straight forward and shown in Figure 8c. The reference *HP* to the new container needs to be stored and the child container flag is updated to `c=01`. The remaining memory section of the embedded container is now vacant and the trailing bytes of the array are shifted to close the gap. This creates new vacant memory at the end of the stream, which is zero initialized. This is required by the scan algorithm to reliably identify invalid nodes. Finally, the *free* attribute of the container head is updated. For efficiency reasons this occasionally triggers a reallocation of the container's memory to keep the unused *free* memory small. Embedded containers can be nested and therefore behave like an ordinary container.

The fourth state `c=11` encodes a child as *path-compressed node (PC)*. Since keys in a trie can share a common prefix, they also can have a unique suffix and encoding a long unique suffix with recursively embedded containers is inefficient. A PC has a one byte header with 7 bits denoting its size and one bit declaring whether a value is attached to the node, limiting the size of a PC to 127 characters. In case a value is stored, it is appended to the PC header. Then the partial suffix key is appended as a regular string. Eventually, an S-Node with a PC encoded child might have to handle another key, which means that parts of the PC node's formerly unique suffix is not unique anymore. In this case it is recursively transformed into an embedded child until the two partial keys can be stored as separate PC nodes.

Operations: The lookup and insertion procedures correspond to a pre-order traversal of the node's trie data structure. Offsets to the next T/S-Node are calculated based on the flags presented in Figure 5 and the size fields of embedded containers or path compressed child nodes. Consequently, the exact-fit approach causes a computational overhead compared to fixed size node elements. Inserts are performed in an order-preserving manner. This requires shifting byte array segments as, e.g., explained in Figure 8. However, maintaining order among siblings has several advantages. Not only are missing keys detected earlier and range queries much faster, it also facilitates several of the performance and efficiency enhancements presented later in Section 3.3.

The complexity of *update* and *delete* operations is comparable to that of insertions or lookups. Deletions will almost always trigger memory shifts, while updates only trigger shifts if a value is added to an existing key without an attached value. Such an operation marks the transition of a type `10` node to type `11`.

Ordered *range queries* use a callback approach. A callback function is passed to the range query function along with a given prefix key. Hyperion invokes the callback function for every key greater than or equal to the provided prefix key, until the callback function returns zero to quit the operation. The stored value is a parameter of the callback function.

At this point we have an understanding of the partial key handling within containers, the different types of containers and how they can be linked to form a trie data structure. Next, we are going to investigate the way these containers are mapped into memory efficiently before examining performance and memory efficiency optimizations in Section 3.3.

3.2 Memory Management

Over-allocation and heap fragmentation considerably influence the memory efficiency of data structures.

Over-allocation refers to allocated but unused memory inside a data structure. ART's Node48, e.g., allocates memory for 48 entries as soon as a node's population exceeds 16 entries, leaving up to 31 entries unused. Over-allocation is typically tolerated to improve performance, e.g., to optimze cache line efficiency or to support SIMD operations. Examples are Judy's *uncompressed node*, ART's *Node256*, and HAT's *array nodes*. Their layout allows calculating an element's offset within the data structure based on its value.

The performance of exact-fit strategies significantly suffers from frequent memory reallocations. Dynamic memory used by exact-fit strategies is either allocated on the heap or by anonymous memory mappings (`mmap`). Heap allocations are generally very fast, but frequent reallocations can lead to excessive heap fragmentation. Memory mappings always include a trap into kernel mode but do not suffer from external fragmentation, as they are always page aligned. Once freed, memory is always returned to the OS.

Hyperion imposes several challenges on the memory subsystem as containers grow in 32 byte increments to minimize over-allocation. These allocations could cause extreme fragmentation if the heap was used. Actually, a first implementation using the *glibc* allocator (based on *ptmalloc2* [23]) collapsed when a Hyperion trie with one billion 32 byte sized containers started to increment its containers. The memory manager must therefore prevent such excessive heap fragmentation.

State-of-the-art memory managers, such as *tcmalloc* [22] or *jemalloc* [20], try to reduce the impact of fragmentation, e.g., by supporting size-specific allocations or by coalescing unused segments. Yet, tcmalloc was designed to improve performance and never releases memory to the operating system; and jemalloc mostly emphasizes on scalable concurrency support.

The unusual but also very predictable memory allocation pattern of our trie encouraged us to tailor a memory manager to our use case, which helped us to overcome heap fragmentation **and** considerably reduce the metadata overhead. It acts as a middleware between our trie and the system's memory management. Small allocations of up to 2,016 bytes are grouped by size and stored in large memory mapped segments. Larger allocations are placed on the heap.

Memory Hierarchy and Hyperion Pointer: The memory manager employs a hierarchical data structure to manage allocated and free segments efficiently. The hierarchy is illustrated in Figure 9. 64 *superbins* are at its top. Superbin SB_0 handles all requests larger than 2,016 bytes; and each superbin SB_i, $i \in [1, 63]$, provides segments of $32 \cdot i$ bytes. Every *superbin* has up to 2^{14} *metabins*, every metabin up to 256 *bins*, and every bin $4,096$ *chunks*. A chunk C_i is the memory segment used to store a trie container. So, every superbin addresses up to 2^{34} chunks.

Instead of eight-byte pointers, our allocator returns five-byte Hyperion Pointers (HPs) containing the IDs of the respective hierarchies. The trie only stores HPs which are resolved by the memory manager. Consequently, the use of HPs completely decouples our trie from the virtual memory. This empowers the memory manager to reorganize and move chunks at will.

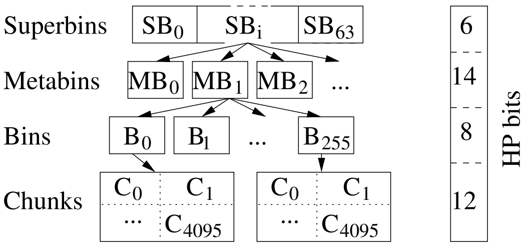


Fig. 9. Memory manager data structure hierarchy

The memory overhead of superbins is small, they fit into a single cache line. The data structure contains a reference to a metabin pointer array, so that new metabins can be initialized individually. Additionally, superbins include a sorted list of 16 non-full metabin IDs to find a free chunk fast. Metabins store, besides housekeeping variables, a 256 bit array to identify non-full bins and an array of the bin structures. Bins use a 4,096 bit array to distinguish used from free chunks, a pointer to the memory mapped segment containing the individual chunks and housekeeping variables. Since each of the 256 bins is 521 bytes large, a metabin consumes a total of 133,416 bytes.

Heap allocators typically store the allocation size internally and impose an eight-byte overhead per segment. In contrast, the kernel does not keep track of memory mapped segment sizes. Therefore, applications must manage segment sizes. As mentioned earlier, the superbin ID defines the chunk size from which the memory mapped segment size can be computed. Hence, our memory savings per allocation accumulate to 11 bytes, compared to heap usage. Once 12,128 out of 1,048,576 chunks are allocated, those savings will compensate for the metabin data structure overhead and **every full metabin saves over 10 MiB**.

We use SIMD instructions to quickly identify free bins and chunks and only issue one kernel trap per 4,098 allocations.

Extended Bins are used for allocations larger than 2,016 bytes and are managed by the superbin SB_0. Extended bins have a size of 16 bytes, as they only store *extended Hyperion Pointers (eHP)*, which contain a regular heap pointer, an integer storing the *requested size* and a short denoting the *over-allocated* memory within that allocation. The remaining two bytes store housekeeping flags. Even though our trie continues to grow in 32 byte steps, eHPs are incremented in intervals of 256 bytes for requests up to 8 KiB, 1 KiB for request up to 16 KiB and 4 KiB otherwise. These larger increments mitigate the effects of heap fragmentation for fast growing containers and improve performance by reducing reallocation overhead.

Chained Extended Bins (CEB) are eight extended bin chunks that are allocated and freed atomically. This means, that a single HP owns eight extended bin chunks, which have to be located in eight consecutive chunks of a bin in SB_0. The heap pointers of some of those extended bins can be void. This concept allows us to access eight individual extended bin chunks without having to handle multiple HPs in our trie. Its application is discussed in the performance and memory efficiency Section 3.3.

Arenas: As mentioned in Section 1, prefix tries are not balanced, and every trie has one fixed root node. This simplifies parallelizing prefix tries. Instead of a single 256-ary trie, an application can create 256 tries T_i with $i \in [0, 255]$. Operations regarding a key k are then mapped to T_{k_0}. Arenas enable thread-safe concurrent access using one memory manager per arena. Every arena therefore has its own set of superbins and controls access using a spin lock. The individual tries T_i are mapped to a set of arenas A_j in a simple round-robin fashion: $T_i \longrightarrow A_{i \bmod j}$.

The concept of arenas is already fully implemented within Hyperion; preliminary evaluations showed that it is necessary to better couple arenas with the memory manager and caching schemes to enable performance improvements beyond small factors of two to three. Furthermore, skewed distributions can limit the benefit of arenas, as only few of them might be filled.

3.3 Performance and Memory Efficiency

The previous sections explained how to build a Hyperion trie and map it into memory. Now we examine four performance and memory efficiency features in more detail. The first improves memory efficiency by *delta encoding* the T/S-Node character values. The second and third measures, called *Jump Successor* and *Jump Tables*, accelerate key lookups by reducing the number of node comparisons during a container traversal. Finally, the *split container* technique decreases shifting overhead by splitting large containers vertically.

Delta Encoding (d): Maintaining a *less than* order among siblings facilitates *delta encoding* of key characters. Figure 10b shows the containers *C3* and *C3** with delta encoded key representations. In *C3* t is the only child of a and has the value 116, but e is a sibling of a and $\Delta(a, e) = 101 - 97 = 4$. Since three bits are sufficient to binary represent 4, this delta can be stored in the delta field *'d'* of the T/S-Nodes and the trailing character byte is unnecessary. In *C3** the t is delta encoded as $\Delta(e, t) = 116 - 101 = 15$, as it is a sibling of the e character. This technique is particularly effective for dense data sets like sequential integers or skewed distributions such as alphanumeric strings. The delta encoding feature comes without memory overhead, as the three bit delta field would be padded otherwise.

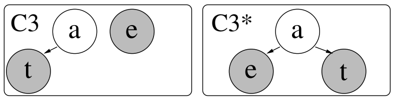
Jump Successor (js): The linear pre-order representation of the container's two-level internal tries leads to $O(k^2)$ comparisons when scanning a trie of degree k. For this reason, we allow the algorithm to jump from one T-Node to the next if the trie becomes too large, and thus to reduce the number of comparisons to at most *2k*. We use the *js*-flag to indicate whether a jump reference is appended to a T-Node. This reference is the offset to the successor sibling in bytes. As the reference is limited to 16 bits, the maximum jump distance is 65,536 bytes and the maximum S-Node size is 256 bytes.

Successor jumps trade capacity for performance. They should only be used if the number of children reaches a (configurable) threshold, where the default value is 2. The default value avoids jumps on nodes with a single child node, because the child node is most likely already located in the current cache line. Therefore, the scanning overhead would be rather small and not worth the two bytes memory spent. A minor drawback of this offset based jump approach is the necessity of updating the offset on insertions or deletions applied on the T-Node's children.

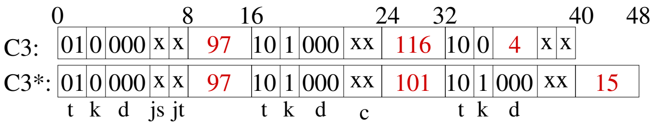
Jump Tables (jt): Successor jumps improve performance, but scanning the trie is still slow for large k. The reason is that the jump distance from a T-Node to its successor also grows with k and the probability that the hardware prefetcher cannot keep up increases, which also increases the cache miss rate. We therefore introduce *jump tables* for T-Nodes and containers. A jump table is a reference list that allows the scanning procedure to jump closer to its target and thus to skip the majority of scans. Both, the container header and the T-Node set their $\mathcal{J}$- or *jt*-flag to indicate the presence of a jump table.

The **T-Node jump table** reduces the latency to access its S-Nodes by storing an array of unsigned shorts referencing 15 of its S-Nodes. Delta encoding is in this case a challenge for our jump table. Jumping from a T-Node to an S-Node, we do not know its predecessor key and are not able to translate the delta to its key character. Therefore, the target key must be known in advance. This can be achieved by storing the target key in the jump table or by jumping to predefined nodes. We chose the second approach.

A T-Node jump table has 15 entries e_i, $i \in \{0, 1, \ldots, 14\}$, where e_i references the S-Node storing the 8-bit partial key



(a) Graph representation


0
8
16
24
32
40
48
C3:
01 0 000 x x 97 10 1 000 xx 116 10 0 4 x x
C3*:
01 0 000 x x 97 10 1 000 xx 101 10 1 000 xx 15
t k d js jt t k d c t k d

(b) Byte array representation with correct key deltas

Fig. 10. Two sample tries illustrating delta encoding

$16 \cdot (i + 1)$. The mapping makes it unnecessary to store the destination key along with the jump offset and the jump table entry for a key k_i can be determined by a bit shift $JT(k_i) = (k_i \gg 4) - 1$. Nevertheless, additional destination points must be created if keys are missing.

The **container jump table** targets T-Nodes instead of S-Nodes. The three bits of the container's jump field *'J'* allows it to grow up to 49 entries in seven entry intervals. A balanced container jump table with 49 entries ensures that at most $\lceil \frac{256}{49} \rceil = 6$ T-Nodes are traversed during lookup. Our container traversal algorithm increases or rebalances the jump table, once eight T-Nodes have been seen. Therefore, once a container jump table has grown to its full extend and its entries are properly balanced, the algorithm will not have to update it again. Special instructions to check for the state of the jump table are unnecessary, which is particularly important, as these would add branch instructions to the critical path of the scanning algorithm.

Each entry is encoded into a 32 bit integer with 8 bits denoting the entry's key and 24 bits storing the offset. Entries are ordered by their keys and occasionally rebalanced. A jump table entry lookup is performed by linearly scanning the entries and identifying the largest entry with a key less than or equal to the required key.

Splitting Containers *vertically* is a performance optimization mechanism. Consider a fully populated container referencing 65,536 child containers. Such a container is over 400 KiB large. Until this size is reached, the shifting and reallocation overhead is considerable. Splitting a container reduces the severity of this overhead.

A container may be split up into a maximum of eight chunks with $chunk_i$ managing the T-Node keys:

$$[32 \cdot i, (32 \cdot (i + 1)) - 1], i \in [0, 7] \tag{3}$$

For example, chunk_0 is responsible for the key range [0-31] and chunk_3 for the range [96-127]. Containers can be split once per iteration. Figure 11 sketches the vertical splitting of container X. Cuts aim to balance the size of the newly created containers. Here the cut is made at the T-Node key 160. The newly created container X_1 contains the key range [0-159] and X_2 the range [160, 255].

Splitting a container results in two newly created containers, each having their own extended bin pointer. Chained pointers introduced in Section 3.2 are used to store extended HPs (eHPs) for all potentially existing eight split containers. The split containers are allocated in consecutive chunks, as indicated by the red border in Figure 11. Although the smallest T-Node in X_1 has the key 57, X_1 is still responsible for the complete range [0-159]. Therefore, the eHP of X_1 is located at the first chained chunk. X_2's eHP has the index 5, because its smallest potential T-Node key is 160 and $\frac{160}{32} = 5$.

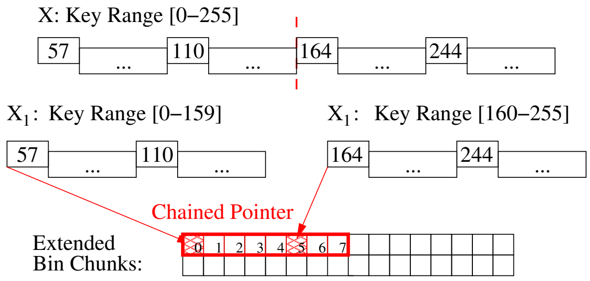


Fig. 11. Container split with chained pointers.

Our scanning algorithm uses the memory manager API to translate HPs into virtual memory addresses. For chained pointers, we add the requested T-Node key to the request. Assume the next partial key we are looking for is 110, which is part of the container X_1. X_1's parent container only stores the HP that directs us to the set of chained pointers. Then the partial key 110 is used to determine, which split container address to return. The chunk index $\lfloor \frac{110}{32} \rfloor = 3$ is the first to check for a valid eHP. As the indices 3, 2 and 1 are all void, the container located at index 0 is returned.

The chained pointer approach facilitates the use of multiple child links without the need of storing multiple links. That would be counter productive, as the HPs are responsible for up to 80% of the inner node's container size.

Finally, we need to understand under which conditions a container is split. Each time a container is scanned during insertion operations our algorithm checks whether:

$$size_c \geq a + b \cdot s \tag{4}$$

where size_c is the container size, $a = 16$ KiB, $b = 64$ KiB and the *split delay* $s \in [0, 3]$. The split delay is the two bit flag S in the container head (c. f. Figure 3) and initialized to zero.

Even though this check might pass, the splitting process might get aborted. The most common case is that the container only stores keys within a single key range (c. f. Eq. 3). This happens for skewed key distributions or already completely split containers. Another case is that either of the two split candidates is smaller than 3 KiB. Then splitting the original container is not very effective. Should the process abort, the containers *split delay S* is incremented. This increases the necessary container size required to pass the initial condition and avoids recurring failing splitting attempts.

3.4 Key Pre-processing

Key pre-processing is a technique to transform keys so that their distribution is more suitable for a data structure. There has been extensive research on key pre-processing for index minimization, especially on compacting tries and modifying the processing order of the key parts [2, 15, 16, 43]. Most

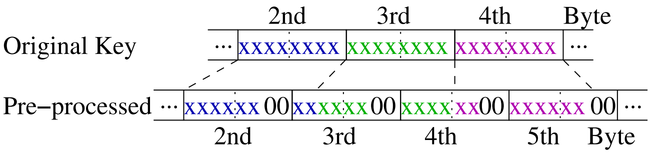

2nd
3rd
4th
Byte
Original Key
··· xxxxxxxx xxxxxxxx xxxxxxxx ···
Pre−processed ··· xxxxxx 00 xxxxxx 00 xxxxxx 00 xxxxxx 00 ···
2nd
3rd
4th
5th
Byte

Fig. 12. Representation of the key transformation.

heuristics and approximations developed are *offline algorithms* which require that the data set is known and finite. But even under these restrictions, several aspects of trie index minimization have been shown to be NP-complete [14, 17].

A key pre-processing heuristic which is *online* in the sense that it does not need complete knowledge of the data in advance, is Oracle's *reverse key index* [41] which reverses a key's byte order and is used to balance indexes with monotonically increasing elements.

More formally, a key pre-processor is an *injective* function f that maps an input key $k \in K_1$ to an output key $k' \in K_2$. *Injectivity* is sufficient to maintain consistency for CRUD operation, but for range queries, f must also be invertible and preserve the *binary-comparable order* $<_{bc}$ among the keys: $\forall k, \ell \in K_1 : k <_{bc} \ell \implies f(k) <_{bc} f(\ell)$.

Hyperion provides an optional key pre-processor that is specifically designed for *uniformly distributed keys* such as random integers or cryptographic hashes which are challenging for many data structures. They result in wide tries with a high number of leaf nodes and therefore decrease the memory efficiency of Hyperion.

Our heuristic overcomes this issue by injecting eight zero-bits (as shown in Figure 12) to reduce the entropy encoded into the first four bytes of the keys. By injecting two zero bits into the second, third and fourth byte of the key, the total number of third level container is reduced from 2^{32} to 2^{26}. Having fewer, but larger containers improves memory efficiency considerably. It is easy to show that the zero-bit injection is an *injective*, *inverse* and *binary-comparable order preserving* function.

We chose to inject the zero bits into the two least significant bits of the respective byte because this preserves the possibility of efficient delta encoding and uniform distribution of the partial key values. With the help of the T-Node jump tables, at most four S-Nodes need to be scanned for the second and fourth byte. The container splits – based on the third byte T-Nodes – are preformed uniformly as well.

Key-preprocessing has to be enabled in advance, but the approach itself does not need a-priori knowledge about the data and can perform the key transformations online. The key size grows by one byte, but due to *path compression*, the memory overhead is low. The effect of this technique is evaluated in Section 4.4.

4 EVALUATION

We performed a comprehensive evaluation of Hyperion, Judy [28], HAT [30], ART [18][35] and HOT [10] and also included the STL map (red-black tree, RB), and the STL unordered_map (hash table). Neither the source code nor the individual configuration parameters of these data structures were modified. All data structures were used as k/v stores.

4.1 Setup

All benchmarks ran on our HPC system, where each node is equipped with two Intel Xeon E5-2630 v4 processors running at 2.2 GHz (3.1 GHz in turbo mode), having 25 MiB L3 cache and 1 TiB of main memory, out of which 978 GiB are usable. The main memory is split into two NUMA nodes of 512 GiB each. The operating system was a 64 bit CentOS 7, and the sources were compiled using GCC-7.2.0 *-O3 -std=gnu99*. The memory consumption was parsed from `/proc/self/status`.

Data Sets: We used integer and string data sets, both evaluated in sequential and randomized order. The integer keys and values were always 64 bit; random integers were generated by the *SIMD-oriented Fast Mersenne Twister* [37].

Variable-length string performance was evaluated using the *Google Books n-gram* corpus [24]. The data set consists of all 1- to 5-grams found in this corpus, including the year a book was published, the number of books it was found in and the total number of occurrences. In our experiments, the n-grams including the year formed the key and the number of books and number of occurrences were encoded as integers and formed the value.

Methodology: We restricted the performance evaluation to *inserts*, *lookups* and *range queries* because the cost of *updates* is equivalent to lookups, and a key *deletion* results in case of Hyperion in a `memmove` within the container so that its runtime is comparable to an insert. For technical reasons, lookups were performed for all values in the same order as they were inserted and range queries were performed from the first element in the data structure to the last one.

Hyperion and all other data structures besides ART and HOT are very similar to a k/v store, where all keys and values are stored in the data structure. ART and HOT instead set pointers to locations outside of the trie. Each lookup therefore requires to follow the pointer and check the requested key and value. This is very efficient if used as a secondary database index, where the key and value are already stored outside of the trie.

We evaluated ART and HOT by adding an array of all k/v pairs. For these pairs, we only accounted for the size of the data without any padding or metadata overhead. This is close to the approach implemented by Alvarez et al. [3]. Leis et. al additionally proposed a *single-value* approach, where each leaf stores one k/v pair [35], which has been implemented in *C* by Dadgar (called *ART_C* in the following) [18]. We performed experiments for ART and ART_C and the original ART has shown to be more efficient, partly because it has not to care about allocating memory for k/v pairs, where the k/v memory consumption is (artificially) minimized.

ART and HOT can be (theoretically) implemented in a way that already stores values up to 8 bytes within the trie, not needing the additional array. This can lead to necessary changes in the node structure and the results, shown as ART_{opt} and HOT_{opt} inside the tables, are therefore lower bounds not covered by experiments. ART_C shows that a general implementation for arbitrary value sizes can easily loose the predicted advantages. We therefore do not add performance values or performance to memory ratios for ART_{opt} and HOT_{opt}.

Judy provides specifically designed versions, where we used *JudyL* for integer keys and *JudySL* for strings.

The threshold for Hyperion's embedded containers was set to 8 KiB for integer keys which means an embedded child container is ejected if the parent container grows larger than 8,192 bytes. For variably sized keys, such as alphanumeric strings, the threshold was set to 16 KiB to better utilize path compression. An embedded container is ejected as soon as it grows beyond its limit of 256 bytes.

As key performance indicators we measure put and get operations in *million operations per second* (MOPS), the duration of range queries (in seconds), the number of bytes per key (*B/key*), and memory consumption with and without payload.

In the interest of easy comparison, we define the ratio *P/M* which includes performance and memory efficiency:

$$\frac{P}{M} = \frac{\text{Puts per sec} + \text{Gets per sec}}{\text{Memory footprint}} \tag{5}$$

Faster put and get operations and a lower memory footprint improve this factor. The P/M ratio is in the following always normalized to the Hyperion value.

4.2 Unlimited Inserts

Since Hyperion is designed to create enormous indexes, we start the evaluation with an experiment that shows how many elements can be indexed within 978 GiB. Figure 13 shows the results for random integers and the English 3-gram string data set. The results already indicate that Hyperion is especially well suited as string k/v store, while still providing a very good memory efficiency for integer keys.

$Hyperion_p$ with *key pre-processing* enabled generates the second largest index for the random integer data set. Hyperion on the other hand outperforms all other data structures by an order of magnitude for sequential strings. To understand these results, we analyze the data structures in the next sections in more detail.

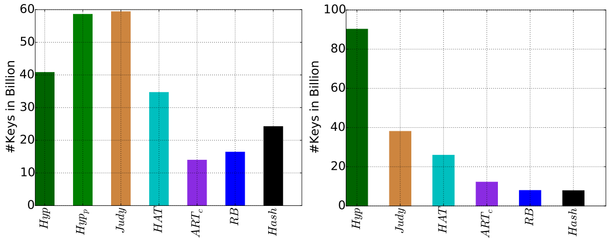


Fig. 13. Memory consumption for the random integer (left) and the 3-gram data set (right).

ART and HOT have not been considered here, as the k/v pre-allocation used in our implementation would have significantly distorted the results.

4.3 String Data Set: Google Books n-grams

For the string data test, we used the English 2-grams `a-d` consisting of 7.948 billion entries. The keys and values have a total size of 167.7 GiB and 59.1 GiB, respectively. The average key size is 22.65 bytes, and keys larger than 255 were skipped, as larger keys are not supported by HOT. Two experiments were run, the first with the n-grams sorted lexicographically enabling a good cache hit rate, the second with the n-grams in random order (see Table 1).

The cache-friendly ordering of the **sequential data set** facilitates high performance inserts and lookups. ART achieves the best insert throughput, closely followed by Judy, which has slightly faster lookups. Hyperion has the lowest memory consumption, while its insert rate is comparable to the hash table's and HAT's. HAT achieves a much higher lookup rate that does not suffer from the rehashing overhead during inserts. Judy's performance leads to the best P/M trade-off, closely followed by Hyperion. HOT's performance is slightly worse than the performance of Judy or ART, while an HOT_{opt} implementation could reduce HOT's memory usage to 101.9 GByte for the sequential string data set, which is only 22% more memory than required by Hyperion.

	7.95 B Sequential String Keys					7.95 B Randomized String Keys				
	Puts	Gets	Mem. GiB		P/M	Puts	Gets	Mem. GiB		P/M
	avg.	MOPS	total	B/key		avg.	MOPS	total	B/key	
Hyperion	0.83	1.10	**85**	11.4	1.0	0.26	0.32	**98**	13.2	**1.0**
JudySL	2.06	**2.77**	202	27.3	**1.0**	0.32	0.33	205	27.7	0.5
HAT	0.83	1.86	281	37.9	0.4	0.33	0.45	281	38.0	0.5
ART_C	2.36	2.70	622	84.1	0.4	0.30	0.32	622	84.1	0.2
ART	**2.37**	2.76	435	58.8	0.5	0.37	0.37	444	59.9	0.3
ART_{opt}			208	28.1				217	29.3	
HOT	1.75	2.46	333	45.0	0.5	0.45	0.53	339	45.7	0.5
HOT_{opt}			106	14.4				112	15.2	
RB-Tree	1.29	1.44	944	127.5	0.1	0.12	0.15	944	127.5	0.0
Hash	0.92	0.85	954	128.8	0.1	0.66	**0.69**	954	128.8	0.2

Table 1. KPI's of the string data sets.

Hyperion memory efficiency significantly benefits from delta encoding, embedded containers and path compression: A memory consumption of 11.27 bytes per key means that Hyperion compresses the average key size of 22.65 bytes to only 3.27 bytes, as 8 bytes are used for the value. The nodes encoded 7,836,534,666 entries as deltas, saving 7.3 GiB memory. Additionally, there have been 953,202,064 embedded containers, where each container saves between 11 bytes and 35 byte, decreasing the memory footprint by at least 10 GByte. 199,706,521 path compressed bytes added around 286 MiB of memory savings.

The results for the **randomized data set** show a considerable performance degradation for most data structures. Especially Judy and ART suffer from the bad caching behavior and are surpassed by HOT and the hash table. Hyperion's performance deteriorates less than Judy's or ART's due to the dense packing in its containers. Encoding 16 bit of the key per container already halves the number of traversed nodes compared to a 256-ary trie. Recursively embedding small child containers into their parents' nodes results in a better cache utilization and facilitates hardware prefetching.

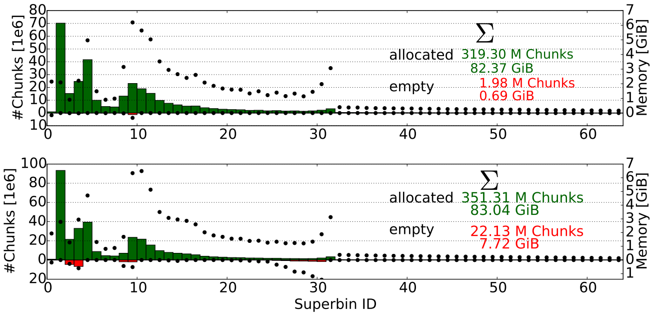


Fig. 14. Hyperion's memory characteristics for the ordered (top) and randomized string data set (bottom).

Hyperion's memory consumption rose by 8.4 GiB, which is caused by memory fragmentation. The x-axis in Figure 14 represents the 64 superbins. The primary y-axis shows the number of chunks, with allocated chunks (green) per superbin growing upwards and empty chunks (red) growing downwards. Empty chunks can be classified as *external fragmentation* and occur, e.g., at the initialization of a new bin. The secondary y-axis sums up the memory consumption of unused and allocated chunks (shown as dots).

Hyperion is by factors more memory-efficient than Judy, ART, and HAT and by orders of magnitude more efficient if we do not include the payload. Consequently, Hyperion dominates the P/M ratio. The normalized P/M ratio of Judy, HOT, and HAT are 0.528, 0.501, and 0.455, respectively. The hash table and ART are outperformed by factors of 4.2 and 6.0 and the tree by an order of magnitude. Nevertheless, a HOT_{opt} implementation could compete with Hyperion concerning memory efficiency and P/M ratio.

4.4 Integer Keys and Values

The next benchmarks use integer keys and values. We inserted 16 billion sequential and 13 billion randomized 64 bit k/v pairs. We reversed the keys' byte order for ART, HAT and Hyperion to adapt to Intel Xeon's *little-endian* format. Otherwise the sequential integers would be disordered. This way they start processing the keys at their most significant byte and fill the trie in a *depth-first* manner. Figure 15 shows the put and get performance and the memory footprint for both data sets, Table 2 summarizes the results.

The **sequential data set** represents the best-case scenario for tries, both with regards to memory efficiency and performance. The first four bytes are highly redundant and only take four different states. In the lower levels of the trie, the nodes are densely populated and the key ordering leads to a cache-friendly access pattern. Such data sets can occur, e.g., if keys are artificially created as unique indexes in databases. The upper right graph in Figure 15 shows the memory consumption of the different tries, where the hatched, light-red part of each bar is equal to the combined memory requirements of the keys and values of 238.4 GByte.

Hyperion and Judy achieve outstanding memory efficiency, which is even smaller than the hatched area. Table 2 shows that Hyperion only requires 1.3 bytes per 8 byte key to index the data set (cf. *'index'* column). The sequential nature allows all Hyperion nodes to delta encode the partial keys, and due to the highly populated containers, memory fragmentation is low. Since JudyL is specifically designed for this key size, it achieves a slightly better memory efficiency and provides the best trade-off between performance and memory efficiency. Hyperion's memory consumption of 9.31 Bytes/key breaks down to the eight byte value, a one byte internal node and 0.31 Bytes/key for the higher level container and other overhead. A Hyperion variant mapping eight byte keys and

	16.0 B Sequential Integer Keys					13.0 B Randomized Integer Keys				
	Puts avg.	Gets MOPS	Mem. total	GiB B/key	P/M	Puts avg.	Gets MOPS	Mem. total	GiB B/key	P/M
Hyperion	1.54	2.96	139	9.3	1.0	0.36	0.64	314	25.9	1.0
$Hyperion_p$						0.41	0.54	**224**	18.5	1.4
JudyL	8.42	21.28	**130**	8.7	**6.3**	1.02	1.07	259	21.4	2.7
HAT	2.99	11.03	359	24.1	0.9	0.55	0.89	334	27.5	1.4
ART_C	9.26	13.34	840	56.4	0.9	0.93	0.92	910	75.1	0.7
ART	**17.32**	25.85	360	24.2	3.9	1.23	0.96	520	43.0	**2.9**
ART_{opt}			121	8.2				327	27.0	
HOT	1.38	2.77	416	27.9	0.3	0.66	0.68	353	29.1	1.3
HOT_{opt}			177	11.9				159	13.1	
RB-Tree	1.34	2.58	954	64.0	0.1	0.23	0.26	775	64.0	0.2
Hash-T	12.24	**34.63**	605	40.6	2.0	1.44	**2.38**	516	42.6	2.4

Table 2. KPI's of the sequential and randomized integer data sets.

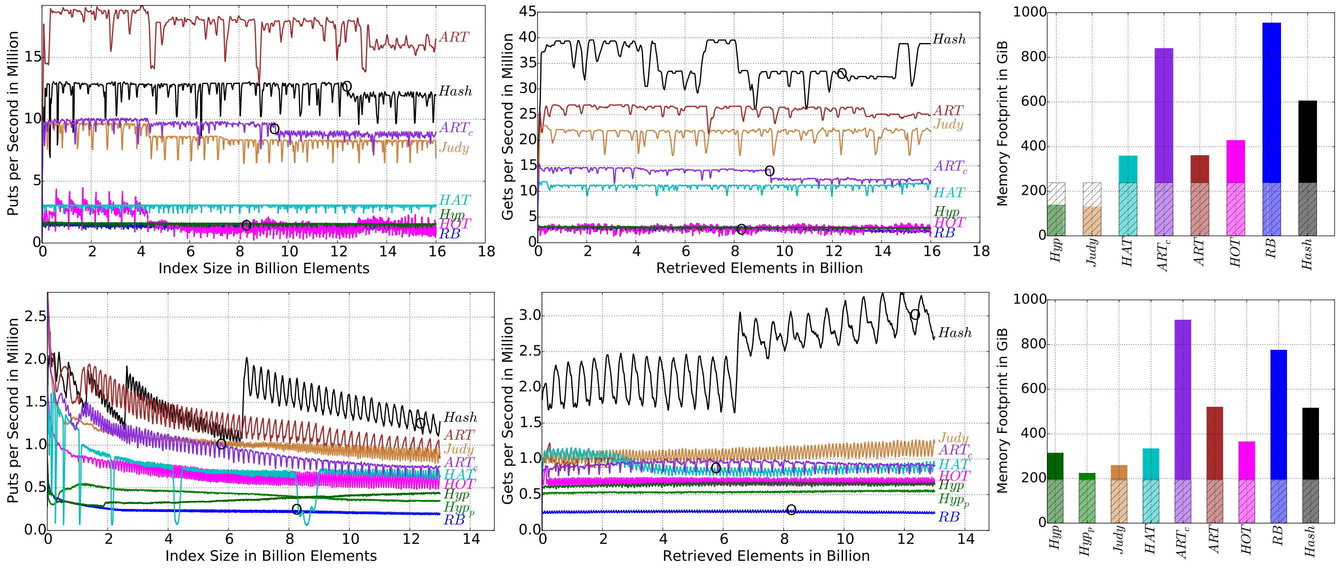


Fig. 15. Results of 16 billion sequential (top) and 13 billion randomized (bottom) 64 bit integer k/v benchmark with put / get operations and memory footprint. Circles mark moments when remote NUMA memory is accessed.

values would not need the one byte internal node and achieve 8.31 Bytes/key efficiency. Also, Hyperion is always faster than the RB-Tree and HOT for sequential integers.

HAT performs lookups much faster than insertions because insertions eventually trigger the resizing of hash containers, where the hash table is doubled in size and the hash values of all keys are recomputed. This overhead also applied to the hash table, but is invisible in the graph, as the x-axis represents the number of index elements, rather than time. The hash table achieves an average of 12.2 MOPS for puts if we ignore the resizing overhead, and 10.8 MOPS otherwise. ART outperforms all other data structures for puts, while still being very fast for gets. The performance of ART_C can be compared with Judy, while it requires much more memory. HOT is unable in this scenario to compete with the rest of the tries, as its memory consumption is in the same order as the hash map, while being significantly slower. ART_{opt} would deliver an excellent memory efficiency, while HOT_{opt} would still need more memory than Hyperion.

The moment when the data structures start to access remote NUMA memory is marked by circles. NUMA effects slightly impact performance for the hash table, ART, ART_C, and HOT.

The **randomized data set** is challenging for all tries. Hyperion's insert rate drops to approximately 290k IOPS during the first 1.8 B inserts and then increases up to 450k IOPS. Constant shifting overhead during random inserts is the root cause for the low throughput. Without *container splitting*, which reduces the container size and thus the amount of shifted memory, the insertion rate would even sink to 90k IOPS until 2 B inserted elements and then grow to roughly 200k IOPS. Hyperion's average lookup rate of 641k IOPS is much faster than the average insertion rate of 359k IOPS, outperforming the RB-tree and being comparable with HOT, while HOT_{opt} would have the best memory consumption.

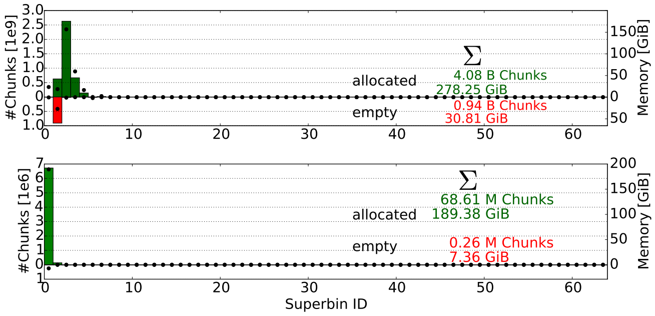


Fig. 16. Hyperion's (top) and $Hyperion_p$'s (bottom) memory usage characteristics. Note the different scales (10^9 vs. 10^6) at the primary Y-axis.

The effects of ART's node structure become visible during the first billion inserts. The two dents represent transitions from *Node16* to *Node48* and from *Node48* to *Node256* on the fourth trie level. Similarly, HAT's increasing re-hashing efforts become visible as the index grows. Hyperion's performance is much improved by activating *key pre-processing*. $Hyperion_p$ reaches a *steady state* after approximately 500 M inserts and has a better insert rate than Hyperion. More importantly, its memory consumption is reduced by 29 %.

Figure 16 shows the allocation distribution for Hyperion and $Hyperion_p$ after 13 billion inserts. Hyperion produced 4.08 B mostly tiny allocations smaller than 128 bytes, totaling to 278.3 GiB and additional 30.8 GiB external fragmentation. Chunk sizes in Hyperion grow in 32 byte intervals and one can expect 16 bytes of internal fragmentation per allocated chunk, leading to 60.8 GiB *internal fragmentation* within this experiment. Extended bins in Hyperion account for 23.5 GiB, and the memory manager's overhead is 1.02 bits per chunk, resulting in 610.4 MiB, leaving 4.5 GiB heap fragmentation.

The motivation for the pre-processing heuristic is to reduce the number of nodes and of allocated chunks in our trie. In fact, applying pre-processing, the number of allocated chunks shrinks by a factor of 72 to 68.87 M. The chunk count reflects one extended bin chunk for each of the 2^{26} possible 4 byte key prefixes. Its higher efficiency originates in the reduction of *internal fragmentation* from 60.8 GiB to 7.36 GiB. Each of its chunks is approximately 3 KiB large. At this size chunks grow in 256 byte increments, which means that one can expect 128 bytes internal fragmentation per chunk. In this experiment it is 117.8 bytes per chunk ($7.36\,GiB/2^{26}$). Figure 16 explains 196.7 out of 223.5 GiB of $Hyperion_p$'s memory consumption. As the memory manager's overhead is only 8.4 MiB, the remaining 26.8 GiB are heap fragmentation.

The random data set did not allow the tries to exploit key prefix redundancies using, e.g., delta encoding as profitably as for the sequential data set. $Hyperion_p$'s memory consumption is therefore only 13.6 % better than Judy's and in general larger than for the sequential data set. On the other hand, the data set is less cache-friendly, which reduces the performance of the other data structures more severely. Again, Judy provides the best trade-off, the hash table the best performance and $Hyperion_p$ the best memory efficiency.

4.5 Range Queries

An advantage of tree-based data structures over hash tables is the efficient range query processing. We evaluate the range query performance for the previously used data sets. HAT's implementation only provides a *begin()* iterator. For a fair comparison, we run a single range query covering the full index. The hash table and ART did not offer an ordered iterator and are not considered.

Hyperion aggregates 16 bits of the key and therefore generates larger containers. This explains the outstanding range query performance shown in Table 3, as the hardware prefetching speeds up the linear scanning procedure. With few exceptions, the other data structures are significantly outperformed. ART_C achieves a good range query performance for sequential data sets while HAT's bad performance was expected because its containers use hash tables to manage their items. An ordered range query requires those items to be sorted first. Judy's results indicate that the nature of the data set has little impact on its generally bad range query performance, while HOT offers a good range query performance for strings, while being comparable to Judy for integers. Overall Hyperion demonstrates extraordinary range query performance, regardless of key type and distribution.

	Integer		String	
	seq.	rand.	seq.	rand.
Hyperion	**166**	2,540	**184**	**205**
$Hyperion_p$		**423**		
Judy	551	2,057	1,661	2,257
HAT	3,823	6,023	3,303	4,103
ART_C	277	2,677	245	1,472
HOT	580		215	
RB-Tree	732	5,920	650	3,752

Table 3. Range query duration in seconds

5 CONCLUSION

This paper introduced Hyperion, an efficient in-memory indexing data structure. We analyzed Hyperion's throughput for puts, gets and range queries and included several data structures for comparison. The data sets used in the evaluation consisted of 64 bit integer key/values and real-life string data. We used sequential and randomly distributed data.

Despite the trend towards vector unit utilization and cache line optimization, we demonstrated that an exact-fit node layout and a linear search approach are still practical in modern in-memory search trees and that especially Hyperion's get performance for non-sequential data sets can compete with all other trie data structures. Hyperion's point furthermore query performance also outperforms a cache-optimized STL `map`. Furthermore, the **memory footprint** is reduced by up to a factor of **2.1 times** and Hyperion processes range queries up to **40% faster** than the best alternative.

Our evaluation examined data structures of exceptional size. Analyzing throughput with reference to index size enables interesting runtime characteristics that were concealed in previous works [3][35][4][5]. We have been able to show, e.g., that previous results on ranking data structures do not hold at this scale [3]. Judy's throughput for random integers, e.g., exceeds ART's after 3 B inserts, while the insert performance of many data structures is highly irregular during the first 1 B operations. Extrapolating average performance to larger indices can therefore result in false assumptions.

Future work will focus on improving Hyperion's multiprocessing performance using Arenas and include extensions towards non-volatile main memory.

REFERENCES

[1] Georgy Adelson-Velsky and Evgenii Landis. 1962. An algorithm for the organization of information. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference (Proceedings of the USSR Academy of Sciences (in Russian))*. Soviet Math. Doklady, 1259 – 1263.

[2] M. Al-Suwaiyel and E Horowitz. 1984. Algorithms for Trie Compaction. *ACM Trans. Database Syst.* 9, 2 (June 1984), 243–263. https://doi.org/10.1145/329.295

[3] Victor Alvarez, Stefan Richter, Xiao Chen, and Jens Dittrich. 2015. A comparison of adaptive radix trees and hash tables. In *31st IEEE International Conference on Data Engineering (ICDE)*. Seoul, South Korea, 1227–1238. https://doi.org/10.1109/ICDE.2015.7113370

[4] Nikolas Askitis and Ranjan Sinha. 2007. HAT-Trie: A Cache-Conscious Trie-Based Data Structure For Strings. In *Proceedings of the Thirtieth Australasian Computer Science Conference (ACSC)*. Ballarat, Victoria, Australia, 97–105. http://crpit.com/abstracts/CRPITV62Askitis.html

[5] Nikolas Askitis and Ranjan Sinha. 2010. Engineering scalable, cache and space efficient tries for strings. *The VLDB Journal* 19, 5 (2010), 633–660. https://doi.org/10.1007/s00778-010-0183-9

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. London, United Kingdom, 53–64. https://doi.org/10.1145/2254756.2254766

[7] Doug Baskins. 2002. A 10-minute description of how Judy arrays work and why they are so fast. http://judy.sourceforge.net/doc/10minutes.htm.

[8] Rudolf Bayer. 1972. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Informatica* 1 (1972), 290–306. https://doi.org/10.1007/BF00289509

[9] Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indices. *Acta Informatica* 1 (1972), 173–189. https://doi.org/10.1007/BF00288683

[10] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD) Conference*. Houston, TX, USA, 521–534. https://doi.org/10.1145/3183713.3196896

[11] Matthias Böhm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner. 2011. Efficient In-Memory Indexing with Generalized Prefix Trees. In *Datenbanksysteme für Business, Technologie und Web (BTW), 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS)*. Kaiserslautern, Germany, 227–246. http://subs.emis.de/LNI/Proceedings/Proceedings180/article22.html

[12] Josiah L Carlson. 2013. *Redis in Action*. Manning Publications.

[13] Chen-Shan Chin, David H Alexander, Patrick Marks, Aaron A Klammer, James Drake, Cheryl Heiner, Alicia Clum, Alex Copeland, John Huddleston, Evan E Eichler, Stephen W Turner, and Jonas Korlach. 2013. Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. *Nature Methods* 10 (2013), 563–569.

[14] Douglas Comer. 1978. The Difficulty of Optimum Index Selection. *ACM Trans. Database Syst.* 3, 4 (Dec. 1978), 440–445. https://doi.org/10.1145/320289.320296

[15] Douglas Comer. 1979. Heuristics for Trie Index Minimization. *ACM Trans. Database Syst.* 4, 3 (Sept. 1979), 383–395. https://doi.org/10.1145/320083.320102

[16] Douglas Comer. 1981. Analysis of a Heuristic for Full Trie Minimization. *ACM Trans. Database Syst.* 6, 3 (Sept. 1981), 513–537. https://doi.org/10.1145/319587.319618

[17] Douglas Comer and Ravi Sethi. 1977. The Complexity of Trie Index Construction. *J. ACM* 24, 3 (July 1977), 428–440. https://doi.org/10.1145/322017.322023

[18] Armon Dadgar. 2017. Adaptive Radix Tree Implementation version 0.9.8. https://github.com/armon/libart.

[19] Rene De La Briandais. 1959. File Searching Using Variable Length Keys. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference (IRE-AIEE-ACM '59 (Western))*. ACM, San Francisco, California, 295–298. https://doi.org/10.1145/1457838.1457895

[20] Jason Evans. 2006. A scalable concurrent malloc(3) implementation for FreeBSD. *(BSDCan - The Technical BSD Conference)*.

[21] Edward Fredkin. 1960. Trie Memory. *Commun. ACM* 3, 9 (Sept. 1960), 490–499. https://doi.org/10.1145/367390.367400

[22] Sanjay Ghemawat and Paul Menage. 2005. TCMalloc: Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[23] Wolfram Gloger. 2006. Ptmalloc. http://www.malloc.de/en.

[24] Google. 2013. Google Books Ngram Dataset Version 2 (CC BY 3.0). http://storage.googleapis.com/books/ngrams/books/datasetsv2.html.

[25] Goetz Graefe and Per-Åke Larson. 2001. B-Tree Indexes and CPU Caches. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)*. Heidelberg, Germany, 349–358. https://doi.org/10.1109/ICDE.2001.914847

[26] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. 2016. LogMine: Fast Pattern Recognition for Log Analytics. In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management (CIKM)*. Indianapolis, IN, USA, 1573–1582. https://doi.org/10.1145/2983323.2983358

[27] Steffen Heinz, Justin Zobel, and Hugh E. Williams. 2002. Burst tries: a fast, efficient data structure for string keys. *ACM Transactions on Information Systems (TOIS)* 20, 2 (2002), 192–223. https://doi.org/10.1145/506309.506312

[28] Hewlett-Packard. 2004. Judy Arrays Implementation version 1.0.5. https://sourceforge.net/projects/judy/.

[29] Thomas N. Hibbard. 1962. Some Combinatorial Properties of Certain Trees With Applications to Searching and Sorting. *J. ACM* 9, 1 (1962), 13–28. https://doi.org/10.1145/321105.321108

[30] Daniel C. Jones. 2017. HAT-trie Implementation version 0.1.2. https://github.com/dcjones/hat-trie.

[31] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur-Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. 2011. Memcached Design on High Performance RDMA Capable Interconnects. In *International Conference on Parallel Processing (ICPP), Taipei, Taiwan, September 13-16*. 743–752.

[32] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Indianapolis, Indiana, USA, 339–350. https://doi.org/10.1145/1807167.1807206

[33] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. 2012. *KISS-Tree*: smart latch-free in-memory indexing on modern architectures. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN)*. Scottsdale, AZ, USA, 16–23. https://doi.org/10.1145/2236584.2236587

[34] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey M. Voas. 2017. DDoS in the IoT: Mirai and Other Botnets. *IEEE Computer* 50, 7 (2017), 80–84. https://doi.org/10.1109/MC.2017.201

[35] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering (ICDE)*. Brisbane, Australia, 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[36] Kurt Maly. 1976. Compressed Tries. *Commun. ACM* 19, 7 (1976), 409–415. https://doi.org/10.1145/360248.360258

[37] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Trans. Model. Comput. Simul.* 8, 1 (Jan. 1998), 3–30. https://doi.org/10.1145/272991.272995

[38] Mikel Izal Daniel Morato, Eduardo Magana, and Santiago Garcia-Jimenez. 2018. Computation of Traffic Time Series for Large Populations of IoT Devices. *Sensors* 19, 1 (2018). https://doi.org/10.3390/s19010078

[39] Donald R. Morrison. 1968. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4 (Oct. 1968), 514–534. https://doi.org/10.1145/321479.321481

[40] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Lombard, IL, USA, April 2-5*. 385–398.

[41] Oracle. 2018. Reverse Key Indexes. https://docs.oracle.com/cloud/latest/db112/CNCPT/indexiot.htm#CBBFDEAJ.

[42] John K. Ousterhout. 1990. Why Aren't Operating Systems Getting Faster As Fast as Hardware?. In *Proceedings of the Usenix Summer 1990 Technical Conference*. Usenix, Anaheim, California, USA, 247–256.

[43] R. Ramesh, A. J. G. Babu, and J. Peter Kincaid. 1989. Variable-depth Trie Index Optimization: Theory and Experimental Results. *ACM Trans. Database Syst.* 14, 1 (March 1989), 41–74. https://doi.org/10.1145/62032.77249

[44] Jun Rao and Kenneth A. Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB)*. Edinburgh, Scotland, UK, 78–89. http://www.vldb.org/conf/1999/P7.pdf

[45] Jun Rao and Kenneth A. Ross. 2000. Making B$^+$-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. ACM, Dallas, Texas, USA, 475–486. https://doi.org/10.1145/342009.335449

[46] William A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News* 23, 1 (1995), 20–24. https://doi.org/10.1145/216585.216588

[47] Yao Yu. 2014. Scaling Redis at Twitter. https://www.youtube.com/watch?v=rP9EKvWt0zo

[48] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. San Francisco, CA, USA, 1567–1581. https://doi.org/10.1145/2882903.2915222